Algorithms Homework#2

201511837 이상민

2.0 Exercise 6 and 7 in page of [E]

Exercise 6

$\boldsymbol{A(n) = 2A(\frac{n}{4}) + \sqrt{n}}$

Master Theorem 에 따라서 r = 2, c = 4, $f(n) = \sqrt{n}$이라 하면, $\log_c r = \frac{1}{2}$이므로, $n^{\log_c r}$과 $f(n)$의 증가 속도가 비슷하므로, $A(n) = O(\sqrt{n}\log n)$이다.

$\boldsymbol{B(n) = 2B(\frac{n}{4}) + n}$

Master Theorem 에 따라서 r = 2, c = 4, $f(n) = n$이라 하면, $\log_c r = \frac{1}{2}$이므로, $n^{\log_c r}$보다 $f(n)$의 증가 속도가 더 빠르므로, $B(n) = O(n)$이다.

$\boldsymbol{C(n) = 2C(\frac{n}{4}) + n^2}$

Master Theorem 에 따라서 r = 2, c = 4, $f(n) = n^2$이라 하면, $\log_c r = \frac{1}{2}$이므로, $n^{\log_c r}$보다 $f(n)$의 증가 속도가 더 빠르므로, $C(n) = O(n^2)$이다.

$\boldsymbol{D(n) = 3D(\frac{n}{3}) + \sqrt{n}}$

Master Theorem 에 따라서 r = 3, c = 3, $f(n) = \sqrt{n}$이라 하면, $\log_c r = 1$이므로, $n^{\log_c r}$보다 $f(n)$의 증가 속도가 더 느리므로, $D(n) = O(n)$이다.

$\boldsymbol{E(n) = 3E(\frac{n}{3}) + n}$

Master Theorem 에 따라서 r = 3, c = 3, $f(n) = n$이라 하면, $\log_c r = 1$이므로, $n^{\log_c r}$보다 $f(n)$의 증가 속도가 더 비슷하므로, $E(n) = O(nlogn)$이다.

$\boldsymbol{F(n) = 3F(\frac{n}{3}) + n^2}$

Master Theorem 에 따라서 r = 3, c = 3, $f(n) = n^2$이라 하면, $\log_c r = 1$이므로, $n^{\log_c r}$보다 $f(n)$의 증가 속도가 더 빠르므로, $F(n) = O(n^2)$이다.

$\boldsymbol{G(n) = 4G(\frac{n}{2}) + \sqrt{n}}$

Master Theorem 에 따라서 r = 4, c = 2, $f(n) = \sqrt{n}$이라 하면, $\log_c r$ = 2이므로, $n^{\log_c r}$보다 $f(n)$의 증가 속도가 더 느리므로, $G(n) = O(n^2)$이다.

$\boldsymbol{H(n) = 4H(\frac{n}{2}) + n}$

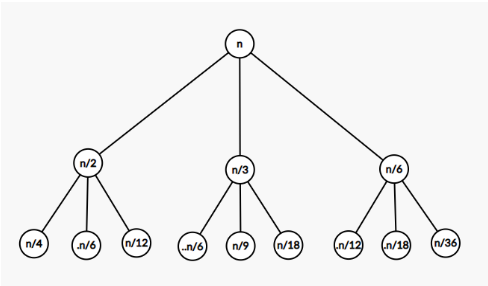
Master Theorem 에 따라서 r = 4, c = 2, $f(n) = \sqrt{n}$이라 하면, $\log_c r$ = 2이므로, $n^{\log_c r}$보다 $f(n)$의 증가 속도가 더 느리므로, $H(n) = O(n^2)$이다.

$\boldsymbol{I(n) = 4I(\frac{n}{2}) + n^2}$

Master Theorem 에 따라서 r = 4, c = 2, $f(n) = n^2$이라 하면, $\log_c r$ = 2이므로, $n^{\log_c r}$보다 $f(n)$의 증가 속도가 더 비슷하므로, $I(n) = O(n^2 \log n)$이다.

Exercise 7.

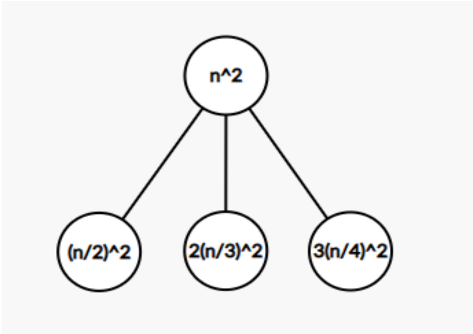
$\boldsymbol{J(n) = J(\frac{n}{2}) + J(\frac{n}{3}) + J(\frac{n}{6}) + n}$



$\frac{n}{2} + \frac{n}{3} + \frac{n}{6} = n,\ \frac{n}{4} + \frac{n}{6} + \frac{n}{12} + \frac{n}{6} + \frac{n}{9} + \frac{n}{18} + \frac{n}{12} + \frac{n}{18} + \frac{n}{36} = n$

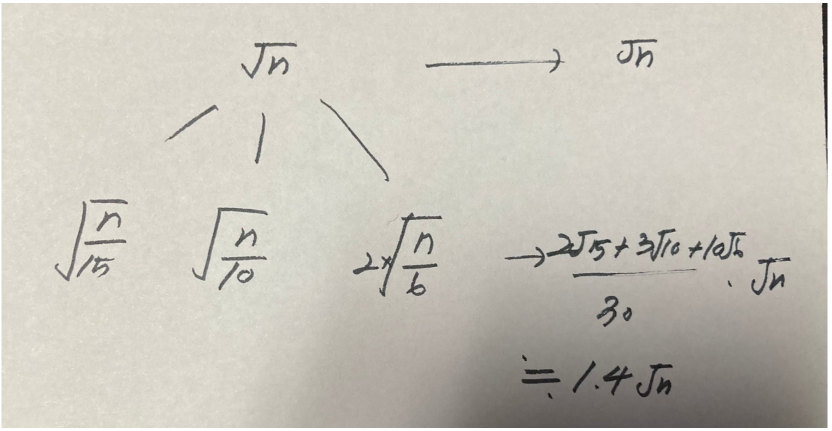
이므로, 연산횟수는 CN 이 되고, 해당 트리의 높이는 $\log n$이므로, $J(n) = O(nlogn)$이다.

$\boldsymbol{K(n) = K(\frac{n}{2}) + 2K(\frac{n}{3}) + 3k(\frac{n}{4}) + n^2}$



$n^2, \frac{95}{144}n^2$,,,,,으로 상수(c)의 값이 수렴하고 있기 때문에 $K(n) = cn^2$임을 알 수 있으며, 이로 인해 $K(n) = O(n^2)$이다.

$\boldsymbol{L(n) = L\left(\frac{n}{15}\right) + L\left(\frac{n}{10}\right) + 2L\left(\frac{n}{6}\right) + \sqrt{n}}$



연산 횟수가 1.4씩 증가하기 때문에 등비수열의 합 공식에 따라서

$\frac{5}{2}(1.4^n - 1)\sqrt{n}$이 나오게 되며, 이로 인해 $L(n) = O(\sqrt{n} * 1.4^n)$이다.

2.1 “정렬후 회전된 배열”이란 (5, 8, 9, 2, 3, 4)와 같은 배열을 말한다. 즉, 정렬이 된 후에 회전 연산이 0 회 이상 적용된 배열이다. 회전 연산이란 배열의 마지막 원소가 처음으로 이동하고 나머지 원소들이 오른쪽으로 한 칸씩 이동하는 것을 말한다. 예를들어, (2, 3, 4, 5, 8, 9)는 정렬된 배열이고 여기에 회전 연산을 1 회 적용하면 (9, 2, 3, 4, 5, 8)이 되고 여기에 회전 연산을 추가로 2 회 적용하면 (5, 8, 9, 2, 3, 4)가 된다. 따라서 (5, 8, 9, 2, 3, 4)는 정렬후 회전된 배열이다.

a. 길이가 n 인 정렬후 회전된 배열 A [0..n-1]가 주어질 때, 이 배열 A 에서 최댓값을 찾는 알고리즘을 설계하고, 분석하시오

```
A = [2, 3, 4, 5]                    # 해당하는 배열


def MaxValFind(A):
    L = len(A)                      # 배열의 길이
    M = L//2                        # 배열의 중간 인덱스 번호
    if L == 2:                      # 계속해서 나누어가면서 최종적으로 배열의 길이가 2 가 되었을 때에는,
        if A[0] >= A[1]:            # 두개의 원소를 비교했을 때, 큰 값을 리턴한다.
            return A[0]
        else:
            return A[1]
    if A[0] >= A[M]:                # 배열 중간의 원소가 배열의 처음의 원소보다 작을때에는,
        return MaxValFind(A[:M])    # 중간 원소를 기준으로 나눈 배열의 앞에 최대값이 존재한다는 반증이기 때문에
    else:                           # 중간 원소를 기준으로 슬라이싱하여 나눈 배열을 재귀를 사용해서 풀어낸다.
        return MaxValFind(A[M:])    # 위와 반대의 경우에는 중간값 뒤의 값들이 포함되어 있는 배열을 사용해서 재귀로 풀어낸다.
                                    # 이와 같이 재귀를 반복해 나가면서 최종적으로 2 개의 원소가 담긴 배열의 원소 값을 비교하였을 때
                                    # 큰 원소의 값을 return 하게 된다면, 해당 결과가 배열의 최대값이다.
print(MaxValFind(A))

# 시간복잡도 : T(n) = T(n/2) + O(1) = O(logn)이다.
# 정확도 : 정렬후 회전된 배열의 어떤 형태를 넣어도 최대값의 원소를 찾을 수 있기 때문에 정확도는 100%라고 볼 수 있다.
```

b. 정렬후 회전된 배열 A [0..n-1]가 회전연산을 몇번 적용한 것인지 알아내는 알고리즘을 설계하고 분석하시오.

```python
A = [4,5,6,8,1,2,3]          # 해당하는 배열
F = 0                        # 배열의 처음 원소의 인덱스 번호
L = len(A)                   # 배열의 총 길이

def NumRotAccount(A, F, L):
    M = (F+L)//2             # 배열의 중간 원소의 인덱스 번호
    # 회전함의 따라 최대 값의 인덱스가 0 부터 +1 씩 증가하므로, 최대값의 인덱스 번호의 +1 이 회전연산의 횟수이다.
    # 하지만, 회전을 안했을 때의 경우에는 0 회전인지, 배열의 길이만큼 회전한건지 알수가 없기 때문에, 생각하지 않았다.
    if L-F == 1:             # 재귀를 반복하면서 나누어진 배열의 길이 - 배열의 처음 원소의 인덱스 번호를 뺀 결과가 인접한 배열의 길이가 2 이라는 것을 의미하며,
        if A[F] >= A[L]:     # 배열의 재귀를 반복하면서, 나누어진 배열의 길이가 2 가 되었으므로,
            return F+1       # 두 개의 원소 중, 더 큰 원소의 인덱스 번호의 +1 의 값을 return 한다.
        else:
            return L+1
    if A[F] >= A[M]:         # 배열의 길이가 2 만큼 나누어지지 않았기 때문에 배열의 길이가 2 가 될 때까지 재귀를 반복한다.
        return NumRotAccount(A, F, M)   # 배열의 처음 원소가 마지막 원소보다 크다면, M 을 기준으로 나누어진 배열의 앞 부분의 배열이 재귀로 넘어간다.
    else:
        return NumRotAccount(A, M, L)   # 위와 반대의 경우, 뒷 부분의 배열이 재귀로 넘어간다.

print(NumRotAccount(A, F, L))

# 시간복잡도 : T(n) = T(n/2) + O(1) = O(logn)이다.
# 정확도 : 배열이 정렬만 되어있을 경우에는, 0 회전인지 배열의 길이만큼 회전한 것인지 알 수 없으므로, 해당 케이스를 제외하고는
# 정렬후 회전된 배열의 회전연산의 횟수를 모두 찾아낼 수 있으므로, 정확도는 100%이다.
```

c. 정렬후 회전된 배열 A [0..n-1]와 k 가 주어질 때, A 안에서 k 를 탐색하는 알고리즘을 설계하고 분석하시오. 즉, A 의 원소 중에 k 가 있으면 그 위치(index)를 출력하고 없으면 -1 을 출력합니다.

```
A = [5, 8, 9, 2, 3, 4]          # 해당하는 배열
k = 8                           # 찾고자하는 k 의 값
I = 0                           # 최대값의 인덱스 번호를 계산하기 위한 변수
def EleIndOfArrFind(A, k, I):
    if(len(A) == 1):                        # 아래의 과정을 반복해서 나온 결과의 배열 길이가 1 이라면,
        if(A[0] == k):                      # 해당 배열의 원소와 찾고자하는 k 의 값을 비교한다.
            print(I)                        # 같다면 계산한 Index 를 출력하고,
        else:                               # 맞지 않다면, -1 을 출력한다.
            print(-1)
    else:                                   # 먼저 1 단계로 찾고자하는 배열을 반으로 슬라이싱을 한다.
        A1 = A[:len(A)//2]
        A2 = A[len(A)//2:]          # 먼저, 배열의 범위에 속하는 지 알아보는 이유는 맨끝의 원소가 맨앞의 원소보다 크다면, 정렬된 배열이라는 반증이기 때문에 해당 과정을 통해 문제를 푼다.
        if(A1[-1] > A1[0]):                 # 잘린 배열 A1 의 뒷부분의 마지막 원소가 잘린 배열의 처음 원소보다 크다면,
            if(k <= A1[-1] and k >= A1[0]):  # 먼저 A1 의 배열의 범위 안에 속하는지 알아보고, 그렇지 않다면 배열 A2 에 k 가 범위에 속하는지 알아본다.
                EleIndOfArrFind(A1, k, I)   # 만약에 A1[-1] > A1[0]을 만족한다면, k 가 A1 의 속하는지 알아낸다.
            else:                           # 속하지 않는다면, A2 에 속하는지 알아내기 위해 A2 를 재귀호출의 매개변수로 넘긴다.
                I += len(A)//2              # 그리고, 배열의 인덱스를 알아내기 위해서 I 의 배열의 값의 반을 더해줌으로써
                EleIndOfArrFind(A2, k, I)   # 인덱스의 번호를 따로 계산한다.
        else:
            if(k <= A2[-1] and k >= A2[0]):  # 이 경우 잘린 A1 배열의 뒷부분의 마지막 원소가 잘린 배열의 처음보다 크지않은 경우이기 때문의 A2 의 범위에 속하는지 먼저 확인을 진행하고,
                I += len(A)//2              # 인덱스의 번호를 계산.
                EleIndOfArrFind(A2, k, I)   # A2 의 범위에 속한다면, A2 를 재귀로 반복한다.
            else:
                EleIndOfArrFind(A1, k, I)   # 그렇지 않다면, A1 을 배열로 사용한다.


EleIndOfArrFind(A, k, I)
# 시간복잡도 : T(n) = T(n/2) + O(1) = O(logn)
# 정확도 : 위의 설명으로 코드를 진행한다면, k 값의 인덱스를 정확하게 찾아낼 수 있으므로 정확도는 100%이다.
```

2.2 입력으로 주어지는 배열 A [0..n-1]은 오름차순으로 정렬되어 있으며 n 개의 서로 다른 정수들을 원소로 가진다. 즉, A[0] < A[1] < … < A[n-1] 이다. 원소들은 양수, 음수 혹은 0 일 수 있다.

a. A[i] = i 를 만족하는 index i 가 존재하는지 알고 싶다.

그런 index i 가 존재하면 찾아서 i 를 출력하고, 없으면 -1 을 출력하는 알고리즘을 설계하고 분석하시오.

```
A = [1,2,3,4,5] #대상이 되는 배열

# A : 배열, F : 배열의 처음 원소의 인덱스 번호, E : 배열의 마지막 원소의 인덱스 번호
def SamePosAndVal(A,F,E):
    M = (F+E) // 2                     # 배열의 중간 원소의 인덱스 번호
    if (A[0] == 0):                    # 배열의 처음이 0 인 경우에, 0 을 return
        return 0
    if (F > E):                        # 배열의 마지막 원소의 인덱스 번호가 배열의 처음 원소의 인덱스 번호보다 작다면,
        return -1                      # 조건에 해당하는 원소가 존재하지 않는다는 반증이기 때문의 -1 을 return 한다.
    if (A[M] < M):                     # 중간 원소의 인덱스 번호보다 A[M]의 값이 더 작다면, 중간 원소의 앞부분에는 찾는 원소가 존재하지 않는다는 반증이기 때문에
        return SamePosAndVal(A,M+1,E)  # 배열의 중간 원소의부터 마지막 원소까지를 재귀호출의 배열로 사용한다.
    elif(A[M] > M):                    # 위의 반대의 경우, 중간 원소의 앞에 해당하는 원소가 있기 때문에 중간 원소를 기준으로 앞의 남아있는 배열을
        return SamePosAndVal(A,F+1,M)  # 다음 재귀호출의 대상으로 삼는다.
    else:
        return M                       # 그리고 나머지의 경우에는 A[M]과 M 의 원소가 같은 경우이기 때문에 조건에 해당하는 원소를 찾은 것이므로 M 을 return 한다.

print(SamePosAndVal(A,0,len(A)-1))

# 시간 복잡도 : T(n) = T(n/2) + O(1) = O(logn).
# 정확성 : 문제의 해당하는 조건을 갖춘 원소들은 모두 해결할 수 있기 때문에 정확성은 100%이다.
```

b. 배열 A 의 원소들이 모두 0 혹은 양수라는 조건이 성립한다면, 위의 문제를 해결하는 더 빠른 알고리즘이 가능하다. 어떻게 할 수 있을까?

```
# 해당하는 배열의 원소 첫번째 인덱스[0]가 원소 0 이 아니라면, 문제의 조건이 서로 다른 양수이기 때문에 뒤에 있는 원소들은 모두 조건을 만족할 수 없다는 반증이 나온다.
# 때문의 시간복잡도는 O(1)이 나오게 되며, a 의 시간 복잡도 보다 빠르게 해결이 가능하다.
```

2.3 최대합 부분배열

길이가 n 인 정수의 배열 A[0..n-1]가 있다. A[a] + A[a+1] +...+ A[b]의 값을 최대화하는 구간 (a,b)를 찾는 방법을 Divide-and-Conquer 전략을 이용하여 설계하고 분석하라.

예를들어, 배열 A 가 아래와 같이 주어졌을 경우 (n= 10),

31 -41 59 26 -53 58 97 -93 -23 84

답은 a = 2, b = 6 인 경우의 59+26-53+58+97=187 가 된다.

```
# 해당 문제를 풀기 위한 중요 접근 방법 : 아래의 경우의 수로 풀 수 있으며, 예외는 없다.
    # 1. [Lower, Mid] : 기준 배열의 왼쪽에 있는 경우
    # 2. [Mid+1, High] : 기준 배열의 오른쪽에 있는 경우
    # 3. 양쪽 모두의 걸쳐 있는 경우

A = [31, -41, 59, 26, -53, 58, 97, -93, -23, 84]        # 대상이 되는 배열

def findMax(arr,start,last):
    def findMidMax(arr,start,mid,last):                 # 왼쪽, 오른쪽에 있는 경우가 아닌 중간을 포함해 걸쳐있는 경우를 위한 함수 선언.
        max_left, max_right = 0, 0                      # 변수를 저장할 임시공간을 선언
        left = -99999                                   # 절대 사용하지 않을 만한 최소값과 비교하기 위한 변수
        right = -99999
        sum = 0                                         # 합계를 비교하기 위해 저장해둘 변수 선언
        for i in range(mid, start-1, -1):               # start 부터 mid 까지의 최대 합을 구한다.
            sum += arr[i]                               # sum 에 나누어진 배열의 값을 더해가며 비교한다.
            if sum > left:                              # 미리 선언 해둔 left 값을 이용하여서 sum 과의 크기 비교에 사용한다.
                left = sum                              # 이렇게 구해진 최대값을 left 값으로 넘기고
                max_left = i                            # 해당 원소의 인덱스 번호를 따로 저장한다.
        sum = 0                                         # 위와 비슷한 방식을 진행하기 위해서 sum 변수를 0 으로 초기화후 다시 사용한다.
        for j in range(mid+1, last+1):                  # Mid 부터 last 까지의 최대 합을 구한다.
            sum += arr[j]                               # 해당 for 문에서는 오른쪽으로 나누어진 배열의 가장 큰 원소의 값과 인덱스를 찾는다.
            if sum > right:                             #
                right = sum                             #
```

```
                max_right = j                           # 
        return (max_left, max_right, left+right)        # 그리고, 해당 부분에서 왼쪽과 오른쪽의 최대 값의 인덱스 번호와 왼쪽과 오른쪽의 최대값을 합을 리턴한다.

    if (start == last):                                 # 배열을 끝까지 나누어서 길이가 1 이 되는 경우,
        return (arr[start],start,last)                  # 배열의 원소값과 인덱스 번호의 시작과 끝을 return 한다.
    else:
        mid = (start+last) // 2                                             # 배열의 중간이 되는 원소의 인덱스 번호
        leftSum, leftStart, leftLast = findMax(arr, start, mid)             # 왼쪽 배열을 계속해서 나누어가면서 길이가 1 이 될 때까지 재귀호출한다.
        rightSum, rightStart, rightLast = findMax(arr, mid+1, last)         # 오른쪽 배열을 계속해서 나누어가면서 길이가 1 이 될 때까지 재귀호출한다.
        midsumStart, midsumLast, midsum = findMidMax(arr, start, mid, last) # 배열의 중간을 기준으로 양쪽 모두의 걸쳐 있는 경우 findMidMax 함수를
                                                                            # 호출하여 앞서 설명한 함수를 재귀호출한다.
    if leftSum >= rightSum and leftSum >= midsum:       # 왼쪽의 최대합이 오른쪽의 최대합과, 중간의 최대합보다 크다면,
        return (leftSum, leftStart, leftLast)           # 왼쪽의 최대합, 시작 인덱스 번호, 끝 인덱스 번호를 return 한다.
    elif (rightSum >= leftSum and rightSum >= midsum):  # 이 과정은 위와 같은 과정이지만,
        return (rightSum,rightStart, rightLast)         # 오른쪽의 최대합이 왼쪽의 최대합과, 중간의 최대합보다 더 큰 경우이다.
    else:                                               # 그리고 나머지의 경우는 중간의 최대합이 더 큰 경우이기 때문에
        return (midsum,midsumStart,midsumLast)          # 중간의 최대합, 중간 배열의 인덱스 시작 번호와 끝 번호를 return 하며, 종료한다.


mm = findMax(A, 0, len(A)-1)
print("a:", mm[1], "b:", mm[2], "최대 합:", mm[0])

# 시간 복잡도 : T(n) = 2T(n/2) + O(n) = O(nlogn)
# 정확도 : 문제에서 제시되는 case 의 배열을 모두 잘 해결하고, 다른 배열에 대해서도 잘 해결할 수 있기 때문에 정확도는 100%이다.
```

문제에 대해서 임한민(201511849), 정범식(201511857), 한상준(201511871) 학우와 함께 토의를 진행하고, 풀이를 진행하였습니다.